JARINGAN SARAF TIRUAN UNTUK MEMPREDIKSI TINGKAT PEMAHAMAN SISIWA TERHADAP MATA PELAJARAN DENGAN MENGGUNAKAN ALGORITMA BACKPROPAGATION

KELOMPOK 6

- Hunafa Rayya
- Ali Abdurrahman Hakim
- Muhammad Muhibuddin Mukhlish

Daftar Isi

Apa Itu Jaringan Saraf Tiruan?

Jaringan saraf tiruan (JST) (artificial neural network (ANN) / simulated neural network (SNN) / neural network(NN)) adalah jaringan yang terdiri atas sekelompok unit pemroses kecil yang dimodelkan berdasarkan jaringan saraf manusia. (Sari Indah Anatta Setiawan, 2011).

Jaringan saraf tiruan adalah paradigma pemrosesan suatu informasi yang terinspirasi oleh sistem sel saraf bilogin sama seperti otak yang memproses suatu informasi.

Pengertian Feedforward

Feedforward Neural Network adalah salah satu jenis JST yang memproses data hanya dalam satu arah, dari input ke output, tanpa adanya siklus atau umpan balik. Jaringan ini dikenal sederhana namun efektif, dengan lapisan-lapisan yang dirancang untuk mentransformasi data masukan secara bertahap hingga menghasilkan output. Bobot antar neuron di jaringan ini diperbarui selama pelatihan untuk meningkatkan akurasi prediksi. Struktur ini cocok untuk berbagai aplikasi, terutama yang tidak memerlukan data berurutan.

Apa Itu Backpropagation?

Backpropagation merupakan metode yang banyak digunakan untuk menyelesaikan suatu permasalahan yang berkaitan dengan prediksi, identifikasi, dan pengenalan pola.

Algoritma ini termasuk dalam supervised learning dimana ciri dari metode ini adalah meminimalkan error pada output yang dihasilkan oleh jaringan.

Algoritma backpropagation untuk neural network umumnya diterapkan pada jaringan berlapis banyak (multilayer).

Bagian-Bagian Backpropagation

Algoritma ini paling tidak mempunyai bagian input, bagian output dan beberapa lapis yang berada di antara input dan output.

Lapis di tengah ini, yang juga dikenal dengan lapis tersembunyi (hidden layer), bisa satu, dua, tiga dst.

Output lapis terakhir dari hidden layer langsung dipakai sebagai output dari neural network.

Fase-Fase pada Backpropagation

1. Fase Propagasi Maju (Feedforward):

- Data input (xi) masuk ke jaringan
- Data mengalir melalui hidden layer (zj)
- Setiap neuron melakukan perhitungan dan meneruskan hasilnya
- Akhirnya menghasilkan output (yk)

2. Fase Propagasi Balik (Backpropagation):

- Output (yk, k=1,2,3,...,m) dibandingkan dengan target yang diinginkan
- Dihitung selisih (error) antara output dan target
- Error ini "dipropagasi balik" ke layer-layer sebelumnya
- Setiap neuron menghitung kontribusinya terhadap error

3. Fase Update Bobot:

- Berdasarkan error yang dihitung, bobot antar neuron disesuaikan
- Penyesuaian bertujuan memperkecil error di iterasi berikutnya
- Proses diulang sampai error cukup kecil atau mencapai batas iterasi

Langkah-Langkah Backpropagation

Langkah 0 :

- Inisialisasi semua bobot dengan bilangan acak kecil
- Menetapkan maksimum epoh, target errordan learningrate
- Inisialisasi, epoh = 0
- Selama epoh < maksimum_epoh dan MSE < target_error, maka akan dikerjakan langkah-langkah berikut

Langkah 1 :

- Jika stopping condition masih belum terpenuhi, lakukan Langkah 2-9

Langkah 2 :

- Untuk setiap data training, lakukan langkah 3-8

Langkah-Langkah Backpropagation

Fase I (Feed forward)

Langkah 3 :

- Setiap unit input menerima sinyal input dan meyebarkan sinyal tersebut pada seluruh unit pada hidden layer

Langkah 4:

- Nilai input pada tiap-tiap unit hidden neuron akan dihitung menggunakan nilai bobotnya:

$$Z\_net_j = V_{0j} + \sum_{i=1}^{n} X_i V_{ij}$$

- Dan memakai fungsi aktivasi sigmoid yang telah ditentukan untuk menghitung sinyal output dari hidden unityang bersangkutan,
- Lalu mengirim sinyal output ke seluruh unit pada unit output.

Keterangan :

z_net_j = sinyal *input* pada *hidden layer* ke – j
v_{jo} = bias ke *hidden layer* ke – j
v_{ji} = bobot antara *unit input layer* ke – I dan *hidden layer* ke - j
x_i = *unit input layer* ke – i
z_j = *unit input layer* ke – j
i = urutan *unit input layer*
j = urutan *unit hidden layer*
p = jumlah *maksimum unit* pada *hidden layer*

Langkah-Langkah Backpropagation

Fase I (Feed forward)

Langkah 5 :

- Setiap unit output (yk ,k=1,2,3,...,m) akan menjumlahkan sinyal-sinyal input yang sudah berbobot termasuk biasnya.

$$y\_net_k = w_{ko} + \sum_{j=1}^{p} z_j w_{kj}$$

- Dan memakai fungsi aktivasi yang telah ditentukan untuk menghitung sinyal output dari unit output yang bersangkutan.

$$y_k = f(y_{net_k}) = \frac{1}{1+e^{-y\_net_k}}$$

Keterangan :

y_net_k = sinyal masukan *output* ke - k
w_{ko} = bias ke *hidden layer* le – k
w_{kj} = *output* ke – k dan *hidden layer* ke – j
z_j = aktivasi *hidden layer* ke – j

Langkah-Langkah Backpropagation

Fase II (Back Propagation)

Langkah 6 :

- Setiap unit output (yk , k=1,2,3,…,m) menerima suatu target (output yang diharapkan) yang akan dibandingkan dengan output yang dihasilkan.

- Faktor δk digunakan untuk menghitung koreksi error (Δwkj) yang dipakai untuk memperbarui wkj, dimana:

$$\delta_k = (t_k - y_k) f'(y_{net_k}) = (t_k - y_k)\, y_k\, (1 - y_k)$$

- Faktor δk ini kemudian dikirim ke layer depannya.

Keterangan :

δ_k = faktor koreksi *error* bobot w_{jk}

t_k = target *output* ke – k

y_k = aktivasi *output* ke – k

Δw_{kj} = nilai koreksi *error* bobot w_{kj}

z_j = aktivasi *hidden layer* ke – j

Langkah-Langkah Backpropagation

Fase II (Back Propagation)

Langkah 7 :

- Setiap hidden unit (Zj,1,2,3,...,p) menjumlah input delta(yang dikirim pada layer pada Langkah 6) yang sudah berbobot.

$$\delta\_net_j = \sum_{k=1}^{m} \delta_k w_{kj}$$

- Kemudian hasilnya akan dikalikan dengan turunan dari fungsi aktivasi yang digunakan jaringan untuk menghasilkan faktor koreksi eror δj, dimana :

$$\delta_{j\,=}\, \delta\_net_j\, f'\left(z_{net\,j}\right) = d_{net\,j} z_j (1 - z_j)$$

- Faktor δj digunakan untuk menghitung koreksi error (Δvji) yang akan dipakai untuk memperbarui vik, dimana :

$$\Delta v_{ji\,=}\, \alpha \delta_k x_i$$

Keterangan :

δ_net_j = jumlah delta bobot *hidden layer* ke – j
δ_k = faktor koreksi *error* bobot w_{kj}
w_{kj} = bobot antara *output* ke – k dan *hidden layer* ke – j
δ_j = faktor koreksi bobot v_{ij}
z_j = aktivasi *hidden layer* ke – j
v_{ji} = nilai koreksi *error* bobot v_{ji}
α = laju percepatan (*learning rate*)
δ_i = faktor koreksi *error* bobot v_{ji}
x_i = unit *input* ke – i

Langkah-Langkah Backpropagation

Fase III (Perubahan Bobot)

Langkah 8 :

- Setiap unit output (yk,k=1,2,3,...,m) akan memperbarui bias dan bobotnya dengan setiap hidden unit.

$$w_{kj}\ (baru) = w_{kj}\ (lama) + \Delta w_{kj}$$

- Begitu juga dengan setiap hidden unit akan memperbarui bias dan bobotnya denga setiap unit-unit input.

$$v_{ji}\ (baru) = v_{ji}\ (lama) + \Delta v_{ji}$$

Langkah 9 :

- Memeriksa stop condition, yaitu jika stop condition telah terpenuhi, maka pelatihan jaringan dapat dihentikan.

Keterangan :

w_{kj} *(baru)* = bobot baru dari unit *hidden layer* menuju unit *output layer*

Δw_{kj}*(lama)* = bobot lama dari unit *hidden layer* menuju unit *output layer*

$\boldsymbol{v_{ji}}$*(baru)* = bobot baru dari unit *hidden layer* menuju unit *output layer*

$\Delta \boldsymbol{v_{ji}}$(lama) = bobot lama dari unit *hidden layer* menuju unit *output layer*

Studi Kasus dan Implementasi Program

NO	NIS	NAMA	VARIABEL INPUT				TARGET
			A	B	C	D	
			X1	X2	X3	X4	X5
1	1514	ADE IRMA CITA DEWI	0.8	0.6	0.6	0.8	1
2	1517	ANDI KUSUMA	0.8	1	0.8	0.8	1
3	1518	ANDREANSYAH	0.8	0.6	0.4	0.4	1
4	1519	ANJAS SULISTIAWAN	0.8	0.8	0.8	1	1
5	1520	APRIL FRANCIS TAMPUBOLON	0.6	0.6	0.8	0.6	1
6	1522	ARYA DWI SUMANTHA SIDAURU	0.8	0.8	0.8	0.8	1
7	1523	ARYA SUKMA JAYA WARDANA	0.6	0.8	0.6	0.8	1
8	1524	DEVI GUSTIADANI PARANGIN-ANGIN	0.8	0.6	0.6	0.8	1
9	1526	DICKY WAHYUDI	0.8	0.8	0.6	0.8	1
10	1527	DWI SHAFIRA BATUBARA	0.6	0.8	0.6	0.6	1
11	1528	EDI KURNIAWAN	0.8	0.8	0.6	0.8	1
12	1529	FAISAL	0.6	0.4	0.6	0.8	1
13	1530	FARADILA HAFIZAH PARINDURI	0.8	0.8	1	1	1
14	1532	ILLIAS	0.6	0.6	0.8	0.6	1
15	1533	INDAH DESWITA	0.8	1	0.8	0.8	1
16	1534	JENI ARISKA	0.8	0.8	0.8	0.8	1
17	1536	LISNA WATI	0.8	0.8	0.6	0.8	1
18	1537	MILO SANDIKA	0.8	0.6	0.8	1	1
19	1538	MONICA APRILLIA DAMANIK	0.8	0.8	0.8	0.8	1
20	1539	MUHAMMAD BAYU SHOPAN	0.8	0.6	0.8	0.6	1
21	1542	NIA BAZHLINA	0.8	0.8	0.6	0.8	1
22	1544	PANCA IRAWAN	0.6	0.6	0.8	0.6	1
23	1545	PUTRA PANDU KHAIRUN NAZRI	0.6	0.4	0.6	0.8	1
24	1546	PUTRI MAYANG SARI	0.6	0.8	0.8	0.8	1
25	1547	RAHMAD	0.6	0.6	0.8	0.8	1
26	1548	RAMA DONA	0.6	0.8	0.8	0.8	1

INISIALISASI

```python
def __init__(self, input_size=4, hidden_size=2, output_size=1):
    # Initialize network architecture
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    # Initialize learning parameters
    self.learning_rate = 0.1  # Learning rate updated to 0.1
    self.momentum = 0.9

    # Initialize weights with small random values
    np.random.seed(42)
    self.hidden_weights = np.random.normal(0, 0.5, (input_size, hidden_size))
    self.output_weights = np.random.normal(0, 0.5, (hidden_size, output_size))

    # Initialize biases
    self.hidden_bias = np.zeros((1, hidden_size))
    self.output_bias = np.zeros((1, output_size))

    # Initialize momentum terms
    self.hidden_weights_momentum = np.zeros_like(self.hidden_weights)
    self.output_weights_momentum = np.zeros_like(self.output_weights)
```

input_size=4 : Menentukan jumlah neuron di layer input. dan Di sini 4 berarti bahwa data input memiliki 4 fitur (kolom dalam dataset).

hidden_size=2 : Menentukan jumlah neuron di layer tersembunyi. dan Layer tersembunyi adalah bagian dari neural network yang memproses data sebelum menghasilkan output.

output_size=1 : Menentukan jumlah neuron di layer output. dan 1 berarti model akan menghasilkan satu nilai prediksi.

self.input_size: Menyimpan jumlah neuron input yang ditentukan saat membuat objek.

self.hidden_size: Menyimpan jumlah neuron di layer tersembunyi.

self.output_size: Menyimpan jumlah neuron di layer output

learning_rate : Mengontrol seberapa besar perubahan bobot selama pelatihan. dan Jika nilainya terlalu besar, model mungkin gagal mencapai hasil optimal. Jika terlalu kecil, pelatihan akan berjalan lambat.

momentum: Digunakan untuk mempercepat pelatihan dengan memanfaatkan informasi dari langkah sebelumnya. dan Membantu menghindari osilasi dalam perubahan bobot.

Bobot (weights) adalah nilai yang menentukan seberapa kuat hubungan antara neuron.

INISIALISASI

```python
def __init__(self, input_size=4, hidden_size=2, output_size=1):
    # Initialize network architecture
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    # Initialize learning parameters
    self.learning_rate = 0.1  # Learning rate updated to 0.1
    self.momentum = 0.9

    # Initialize weights with small random values
    np.random.seed(42)
    self.hidden_weights = np.random.normal(0, 0.5, (input_size, hidden_size))
    self.output_weights = np.random.normal(0, 0.5, (hidden_size, output_size))

    # Initialize biases
    self.hidden_bias = np.zeros((1, hidden_size))
    self.output_bias = np.zeros((1, output_size))

    # Initialize momentum terms
    self.hidden_weights_momentum = np.zeros_like(self.hidden_weights)
    self.output_weights_momentum = np.zeros_like(self.output_weights)
```

Inisialisasi Bobot: Bobot di layer tersembunyi: Matriks ukuran (input_size x hidden_size) (4x2). dan Bobot di layer output: Matriks ukuran (hidden_size x output_size) (2x1).
Distribusi Normal: Bobot diambil dari distribusi normal dengan rata-rata 0 dan standar deviasi 0.5. Ini memastikan nilai acak kecil sehingga model lebih stabil saat pelatihan.
Bias adalah nilai tambahan yang membantu model mempelajari pola yang lebih kompleks.
np.zeros: Bias di layer tersembunyi: Matriks ukuran (1 x hidden_size) (1x2). dan Bias di layer output: Matriks ukuran (1 x output_size) (1x1).
Momentum: Membantu mengurangi osilasi perubahan bobot dengan menggunakan gradien dari langkah sebelumnya.
Inisialisasi Momentum:
Matriks ukuran sama dengan bobot : Hidden layer: Sama dengan ukuran self.hidden_weights (4x2). dan Output layer: Sama dengan ukuran self.output_weights (2x1).

```
Tabnine | Edit | Test | Explain | Document | Ask
def sigmoid(self, x):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-x))

Tabnine | Edit | Test | Explain | Document | Ask
def sigmoid_derivative(self, x):
    """Derivative of sigmoid function"""
    return x * (1 - x)
```

Sigmoid adalah fungsi aktivasi yang memetakan nilai input ke dalam rentang (0, 1).
Tujuan:
Membuat keluaran jaringan saraf mudah ditafsirkan sebagai probabilitas.
Jika input besar positif → output mendekati 1.
Jika input besar negatif → output mendekati 0.

Jika input x = 2, maka

$$\text{Sigmoid}(2) = \frac{1}{1 + e^{-2}} \approx 0.88$$

Fungsi sigmoid_derivative menghitung turunan dari fungsi sigmoid.
Tujuan:
Membantu dalam backpropagation (pelatihan jaringan saraf) untuk menghitung gradien.
Cara Kerja:
Jika output sigmoid adalah 0.8, maka turunan sigmoid:

Sigmoid Derivative : (0.8)=0.8×(1−0.8)=0.16

```
Tabnine | Edit | Test | Explain | Document | Ask
def forward(self, X):
    """Forward pass through the network"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)

    # Input to hidden layer
    self.hidden_sum = np.dot(X, self.hidden_weights) + self.hidden_bias
    self.hidden_output = self.sigmoid(self.hidden_sum)

    # Hidden to output layer
    self.output_sum = np.dot(self.hidden_output, self.output_weights) + self.output_bias
    self.output = self.sigmoid(self.output_sum)

    return self.output
```

if len(X.shape) == 1:
X = X.reshape(1, -1)

- Input X diperiksa dimensinya.
- Jika X adalah vektor 1D (hanya memiliki satu dimensi), maka diubah menjadi array 2D dengan satu baris dan banyak kolom.

self.hidden_sum = np.dot(X, self.hidden_weights) + self.hidden_bias
self.hidden_output = self.sigmoid(self.hidden_sum)

Input ke Hidden Layer:
Menghitung kombinasi linear dari input X, bobot self.hidden_weights, dan bias self.hidden_bias.

$$\text{hidden\_sum} = X \cdot W_{\text{hidden}} + b_{\text{hidden}}$$

Aktivasi Hidden Layer:
Menggunakan fungsi aktivasi sigmoid untuk mengubah nilai hidden_sum menjadi output non-linear.

$$\text{hidden\_output} = \sigma(\text{hidden\_sum}) = \frac{1}{1 + e^{-\text{hidden\_sum}}}$$

```
Tabnine | Edit | Test | Explain | Document | Ask
def forward(self, X):
    """Forward pass through the network"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)

    # Input to hidden layer
    self.hidden_sum = np.dot(X, self.hidden_weights) + self.hidden_bias
    self.hidden_output = self.sigmoid(self.hidden_sum)

    # Hidden to output layer
    self.output_sum = np.dot(self.hidden_output, self.output_weights) + self.output_bias
    self.output = self.sigmoid(self.output_sum)

    return self.output
```

self.output_sum = np.dot(self.hidden_output, self.output_weights) + self.output_bias
self.output = self.sigmoid(self.output_sum)

Hidden to Output Layer:
Kombinasi linear dari keluaran hidden layer (hidden_output), bobot self.output_weights, dan bias self.output_bias.

$$\text{output\_sum} = \text{hidden\_output} \cdot W_{\text{output}} + b_{\text{output}}$$

Aktivasi Output:
Fungsi sigmoid diterapkan untuk mendapatkan keluaran akhir self.output.

$$\text{output} = \sigma(\text{output\_sum}) = \frac{1}{1 + e^{-\text{output\_sum}}}$$

self.hidden_output adalah keluaran dari layer tersembunyi.

BACKWARD

```
def backward(self, X, y, output):
    """Backward pass to update weights"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Calculate output layer error
    output_error = y - output
    output_delta = output_error * self.sigmoid_derivative(output)

    # Calculate hidden layer error
    hidden_error = np.dot(output_delta, self.output_weights.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

    # Update weights with momentum
    output_weights_update = (self.learning_rate * np.dot(self.hidden_output.T, output_delta) +
                             self.momentum * self.output_weights_momentum)
    hidden_weights_update = (self.learning_rate * np.dot(X.T, hidden_delta) +
                             self.momentum * self.hidden_weights_momentum)

    # Update weights and biases
    self.output_weights += output_weights_update
    self.hidden_weights += hidden_weights_update
    self.hidden_bias += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    self.output_bias += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)

    # Store momentum terms
    self.output_weights_momentum = output_weights_update
    self.hidden_weights_momentum = hidden_weights_update
```

```
if len(X.shape) == 1:
    X = X.reshape(1, -1)
if len(y.shape) == 1:
    y = y.reshape(-1, 1)
```

- Jika data input (X) atau target (y) adalah array satu dimensi, ubah menjadi bentuk dua dimensi.
- Ini dilakukan agar operasi matematika seperti perkalian matriks bisa berjalan dengan benar

output_error:

- Menghitung perbedaan antara target sebenarnya (y) dan prediksi (output).
- Contoh: Jika target adalah 1, tetapi prediksi 0.8, maka error = 1 - 0.8 = 0.2.

output_delta:

- Error ini dikalikan dengan turunan fungsi aktivasi (sigmoid).
- Turunan sigmoid membantu menentukan seberapa besar perubahan yang diperlukan.

BACKWARD

```python
def backward(self, X, y, output):
    """Backward pass to update weights"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Calculate output layer error
    output_error = y - output
    output_delta = output_error * self.sigmoid_derivative(output)

    # Calculate hidden layer error
    hidden_error = np.dot(output_delta, self.output_weights.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

    # Update weights with momentum
    output_weights_update = (self.learning_rate * np.dot(self.hidden_output.T, output_delta) +
                             self.momentum * self.output_weights_momentum)
    hidden_weights_update = (self.learning_rate * np.dot(X.T, hidden_delta) +
                             self.momentum * self.hidden_weights_momentum)

    # Update weights and biases
    self.output_weights += output_weights_update
    self.hidden_weights += hidden_weights_update
    self.hidden_bias += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    self.output_bias += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)

    # Store momentum terms
    self.output_weights_momentum = output_weights_update
    self.hidden_weights_momentum = hidden_weights_update
```

hidden_error:
Error dari layer output diteruskan kembali ke layer tersembunyi menggunakan bobot layer output (self.output_weights).
hidden_delta:
Sama seperti sebelumnya, kita kalikan error ini dengan turunan fungsi sigmoid untuk menghitung perubahan yang dibutuhkan di layer tersembunyi.

self.learning_rate * np.dot(self.hidden_output.T, output_delta):
Ini adalah bagian pembaruan standar tanpa momentum.
np.dot(self.hidden_output.T, output_delta):
Menghitung gradien error untuk layer output.
Gradien ini menunjukkan arah dan besar perubahan yang harus dilakukan pada bobot.
self.learning_rate:
Faktor untuk mengontrol besar langkah pembaruan.

self.momentum * self.output_weights_momentum:

Menambahkan pengaruh dari pembaruan bobot sebelumnya.
self.momentum:
Konstanta momentum (biasanya 0.9), mengatur seberapa besar pengaruh dari iterasi sebelumnya.
self.output_weights_momentum:
Menyimpan pembaruan bobot terakhir dari iterasi sebelumnya.

BACKWARD

```python
def backward(self, X, y, output):
    """Backward pass to update weights"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Calculate output layer error
    output_error = y - output
    output_delta = output_error * self.sigmoid_derivative(output)

    # Calculate hidden layer error
    hidden_error = np.dot(output_delta, self.output_weights.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

    # Update weights with momentum
    output_weights_update = (self.learning_rate * np.dot(self.hidden_output.T, output_delta) +
                             self.momentum * self.output_weights_momentum)
    hidden_weights_update = (self.learning_rate * np.dot(X.T, hidden_delta) +
                             self.momentum * self.hidden_weights_momentum)

    # Update weights and biases
    self.output_weights += output_weights_update
    self.hidden_weights += hidden_weights_update
    self.hidden_bias += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    self.output_bias += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)

    # Store momentum terms
    self.output_weights_momentum = output_weights_update
    self.hidden_weights_momentum = hidden_weights_update
```

self.learning_rate * np.dot(X.T, hidden_delta):
Sama seperti pada output layer, tetapi dilakukan untuk layer tersembunyi.

np.dot(X.T, hidden_delta):
Menghitung gradien error untuk layer tersembunyi.

self.momentum * self.hidden_weights_momentum:
Menambahkan momentum dari perubahan bobot layer tersembunyi pada iterasi sebelumnya.

self.output_weights += output_weights_update:

- Menambahkan perubahan (update) yang sudah dihitung sebelumnya (**output_weights_update**) ke bobot output (self.**output_weights**).
- Memperbarui bobot pada layer output untuk memperbaiki kesalahan prediksi.

self.hidden_weights += hidden_weights_update:

- Menambahkan perubahan (**hidden_weights_update**) ke bobot hidden layer (**self.hidden_weights**).
- Memperbarui bobot pada layer tersembunyi untuk memperbaiki representasi fitur yang dikirim ke layer output.

BACKWARD

```python
def backward(self, X, y, output):
    """Backward pass to update weights"""
    if len(X.shape) == 1:
        X = X.reshape(1, -1)
    if len(y.shape) == 1:
        y = y.reshape(-1, 1)

    # Calculate output layer error
    output_error = y - output
    output_delta = output_error * self.sigmoid_derivative(output)

    # Calculate hidden layer error
    hidden_error = np.dot(output_delta, self.output_weights.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

    # Update weights with momentum
    output_weights_update = (self.learning_rate * np.dot(self.hidden_output.T, output_delta) +
                             self.momentum * self.output_weights_momentum)
    hidden_weights_update = (self.learning_rate * np.dot(X.T, hidden_delta) +
                             self.momentum * self.hidden_weights_momentum)

    # Update weights and biases
    self.output_weights += output_weights_update
    self.hidden_weights += hidden_weights_update
    self.hidden_bias += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    self.output_bias += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)

    # Store momentum terms
    self.output_weights_momentum = output_weights_update
    self.hidden_weights_momentum = hidden_weights_update
```

self.hidden_bias += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True):

- Menambahkan pembaruan pada bias hidden layer.
- Nilai hidden_delta (kesalahan di layer tersembunyi) dijumlahkan untuk semua neuron, lalu dikalikan dengan learning rate untuk menentukan langkah pembaruan.
- Menyesuaikan bias pada hidden layer agar error di layer tersebut berkurang.

self.output_bias += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True):

- Menambahkan pembaruan pada bias output layer.
- output_delta (kesalahan di layer output) dijumlahkan dan dikalikan dengan learning rate.

self.output_weights_momentum = output_weights_update:

- Menyimpan pembaruan terakhir untuk bobot layer output (output_weights_update) ke variabel output_weights_momentum.
- Mempertahankan nilai pembaruan ini agar bisa digunakan pada iterasi berikutnya sebagai momentum.

self.hidden_weights_momentum = hidden_weights_update:

- Menyimpan pembaruan terakhir untuk bobot layer tersembunyi (hidden_weights_update) ke variabel hidden_weights_momentum.
- Sama seperti sebelumnya, tetapi untuk layer tersembunyi.

```
Tabnine | Edit | Test | Explain | Document | Ask
def train(self, X, y, epochs=5000, target_mse=0.0001, batch_size=1):
    """Train the network"""
    X = np.array(X)
    y = np.array(y)

    mse_history = []

    for epoch in range(epochs):
        total_mse = 0

        # Mini-batch training
        for i in range(0, len(X), batch_size):
            batch_X = X[i:i+batch_size]
            batch_y = y[i:i+batch_size]

            output = self.forward(batch_X)
            self.backward(batch_X, batch_y, output)

            # Calculate MSE
            mse = np.mean(np.square(batch_y - output))
            total_mse += mse

        avg_mse = total_mse / (len(X) / batch_size)
        mse_history.append(avg_mse)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, MSE: {avg_mse:.6f}")

        if avg_mse <= target_mse:
            print(f"Target MSE reached at epoch {epoch}")
            break

    return mse_history
```

TRAINING

X = np.array(X) dan **y = np.array(y):**

- Mengubah input dan target menjadi array NumPy untuk memastikan kompatibilitas dengan operasi matematis.

mse_history = []:

- List kosong untuk menyimpan sejarah error (Mean Squared Error, MSE) pada setiap epoch.

Apa itu Epoch?

- Satu epoch adalah ketika seluruh data dilatih satu kali melalui jaringan neural.

for epoch in range(epochs):

- Melakukan pelatihan untuk sejumlah epoch yang ditentukan (default: 5000).

total_mse = 0:

- Menginisialisasi total error (MSE) untuk epoch saat ini.

```
Tabnine | Edit | Test | Explain | Document | Ask
def train(self, X, y, epochs=5000, target_mse=0.0001, batch_size=1):
    """Train the network"""
    X = np.array(X)
    y = np.array(y)

    mse_history = []

    for epoch in range(epochs):
        total_mse = 0

        # Mini-batch training
        for i in range(0, len(X), batch_size):
            batch_X = X[i:i+batch_size]
            batch_y = y[i:i+batch_size]

            output = self.forward(batch_X)
            self.backward(batch_X, batch_y, output)

            # Calculate MSE
            mse = np.mean(np.square(batch_y - output))
            total_mse += mse

        avg_mse = total_mse / (len(X) / batch_size)
        mse_history.append(avg_mse)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, MSE: {avg_mse:.6f}")

        if avg_mse <= target_mse:
            print(f"Target MSE reached at epoch {epoch}")
            break

    return mse_history
```

Apa itu Mini-Batch?

- Data pelatihan dipecah menjadi kelompok kecil (batch) untuk efisiensi.
- **batch_size=1** berarti pelatihan dilakukan satu per satu (stochastic gradient descent).

Langkah Pelatihan dalam Mini-Batch:

- **Ambil Batch:**
- **batch_X**: Subset dari input data.
- **batch_y**: Subset dari target.
- **Forward Pass:**
- **output = self.forward(batch_X):**
- Hitung keluaran jaringan untuk data dalam batch.
- **Backward Pass:**
- **self.backward(batch_X, batch_y, output):**
- Hitung error dan perbarui bobot serta bias berdasarkan data batch.

```
Tabnine | Edit | Test | Explain | Document | Ask
def train(self, X, y, epochs=5000, target_mse=0.0001, batch_size=1):
    """Train the network"""
    X = np.array(X)
    y = np.array(y)

    mse_history = []

    for epoch in range(epochs):
        total_mse = 0

        # Mini-batch training
        for i in range(0, len(X), batch_size):
            batch_X = X[i:i+batch_size]
            batch_y = y[i:i+batch_size]

            output = self.forward(batch_X)
            self.backward(batch_X, batch_y, output)

            # Calculate MSE
            mse = np.mean(np.square(batch_y - output))
            total_mse += mse

        avg_mse = total_mse / (len(X) / batch_size)
        mse_history.append(avg_mse)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, MSE: {avg_mse:.6f}")

        if avg_mse <= target_mse:
            print(f"Target MSE reached at epoch {epoch}")
            break

    return mse_history
```

Mean Squared Error (MSE) adalah rata-rata dari kuadrat selisih antara prediksi dan target

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\text{Target}_i - \text{Output}_i)^2$$

np.square(batch_y - output):

- Menghitung kuadrat error untuk setiap data dalam batch.

np.mean(...):

- Mengambil rata-rata dari semua error di batch.

total_mse += mse:

- Menambahkan error batch ini ke total error epoch.

avg_mse:

- Rata-rata error (MSE) untuk seluruh epoch dihitung dengan membagi **total_mse** dengan jumlah batch.

mse_history.append(avg_mse):

- Simpan MSE epoch ini ke dalam **mse_history** untuk analisis atau visualisasi nanti.

```
Tabnine | Edit | Test | Explain | Document | Ask
def train(self, X, y, epochs=5000, target_mse=0.0001, batch_size=1):
    """Train the network"""
    X = np.array(X)
    y = np.array(y)

    mse_history = []

    for epoch in range(epochs):
        total_mse = 0

        # Mini-batch training
        for i in range(0, len(X), batch_size):
            batch_X = X[i:i+batch_size]
            batch_y = y[i:i+batch_size]

            output = self.forward(batch_X)
            self.backward(batch_X, batch_y, output)

            # Calculate MSE
            mse = np.mean(np.square(batch_y - output))
            total_mse += mse

        avg_mse = total_mse / (len(X) / batch_size)
        mse_history.append(avg_mse)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, MSE: {avg_mse:.6f}")

        if avg_mse <= target_mse:
            print(f"Target MSE reached at epoch {epoch}")
            break

    return mse_history
```

if epoch % 100 == 0:

- Setiap 100 epoch, cetak MSE untuk memantau kemajuan pelatihan.

if avg_mse <= target_mse:

- Jika MSE sudah lebih kecil atau sama dengan target (default: 0.0001), pelatihan dihentikan lebih awal.
- **break:**
- Keluar dari loop epoch.

return mse_history

Mengembalikan daftar **mse_history** yang berisi error pada setiap epoch untuk analisis lebih lanjut, seperti membuat grafik pelatihan.

```python
def predict(self, X):
    """Make predictions using the trained network"""
    return self.forward(X)
```

X:

- Input data yang ingin diprediksi.
- Bisa berupa satu data (vektor) atau beberapa data (matriks).

self.forward(X):

- Fungsi **forward** dipanggil untuk menghitung hasil prediksi dari input X.
- Fungsi **forward** memproses data dari layer input, melalui layer tersembunyi, hingga ke layer output.

return:

- Hasil dari fungsi forward dikembalikan sebagai output prediksi.

```python
# Convert to DataFrame
df = pd.DataFrame(data)

# Split features and target
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values.reshape(-1, 1)

# Normalize data
scaler_X = MinMaxScaler(feature_range=(0.1, 0.8))
scaler_y = MinMaxScaler(feature_range=(0.1, 0.8))

X_normalized = scaler_X.fit_transform(X)
y_normalized = scaler_y.fit_transform(y)

# Create and train model
model = BackpropagationNN(input_size=4, hidden_size=2, output_size=1)
history = model.train(X_normalized, y_normalized, epochs=5000, target_mse=0.00001)

# Predictions and metrics
predictions_normalized = model.predict(X_normalized)
predictions = scaler_y.inverse_transform(predictions_normalized)
mse = np.mean(np.square(y - predictions))
rmse = np.sqrt(mse)

# Accuracy calculation
predicted_classes = [1 if pred >= 0.5 else 0 for pred in predictions.flatten()]
actual_classes = y.flatten()
correct_predictions = np.sum(np.array(predicted_classes) == np.array(actual_classes))
accuracy = (correct_predictions / len(y)) * 100
```

df.iloc[:, :-1]:

- Memilih semua kolom kecuali kolom terakhir dari dataset df.
- X: Berisi fitur-fitur yang digunakan sebagai input untuk jaringan neural.
- Contoh: Jika dataset memiliki 4 kolom (A, B, C, D), maka X akan berisi kolom A, B, dan C.

df.iloc[:, -1]:

- Memilih kolom terakhir dari dataset df.
- y: Berisi nilai target atau label (yang ingin diprediksi).

reshape(-1, 1):

- Mengubah y menjadi array 2D dengan bentuk (n, 1) agar cocok dengan jaringan neural.

Apa Itu Normalisasi?

- Normalisasi mengubah data menjadi rentang tertentu (misalnya, 0.1 hingga 0.8) agar semua fitur memiliki skala yang seragam.
- Membantu jaringan neural berfungsi lebih baik karena data dalam skala yang sama mempermudah pembelajaran.

MinMaxScaler(feature_range=(0.1, 0.8)):

- Membuat objek scaler untuk mengubah data menjadi rentang 0.1 hingga 0.8.
- scaler_X untuk fitur (X).
- scaler_y untuk target (y).

fit_transform:

- **scaler_X.fit_transform(X):**
- Menghitung parameter normalisasi dari X dan menerapkannya.
- **scaler_y.fit_transform(y):**
- Menghitung parameter normalisasi dari X dan menerapkannya.

```python
# Convert to DataFrame
df = pd.DataFrame(data)

# Split features and target
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values.reshape(-1, 1)

# Normalize data
scaler_X = MinMaxScaler(feature_range=(0.1, 0.8))
scaler_y = MinMaxScaler(feature_range=(0.1, 0.8))

X_normalized = scaler_X.fit_transform(X)
y_normalized = scaler_y.fit_transform(y)

# Create and train model
model = BackpropagationNN(input_size=4, hidden_size=2, output_size=1)
history = model.train(X_normalized, y_normalized, epochs=5000, target_mse=0.00001)

# Predictions and metrics
predictions_normalized = model.predict(X_normalized)
predictions = scaler_y.inverse_transform(predictions_normalized)
mse = np.mean(np.square(y - predictions))
rmse = np.sqrt(mse)

# Accuracy calculation
predicted_classes = [1 if pred >= 0.5 else 0 for pred in predictions.flatten()]
actual_classes = y.flatten()
correct_predictions = np.sum(np.array(predicted_classes) == np.array(actual_classes))
accuracy = (correct_predictions / len(y)) * 100
```

BackpropagationNN(input_size=4, hidden_size=2, output_size=1):

- Membuat model jaringan neural dengan:
- 4 neuron input.
- 2 neuron di layer tersembunyi.
- 1 neuron output.

model.train(X_normalized, y_normalized, epochs=5000, target_mse=0.00001):

- Melatih jaringan neural:
- Menggunakan data yang telah dinormalisasi.
- Selama maksimum 5000 epoch.
- Dengan target error (Mean Squared Error, MSE) sebesar 0.00001.

history:

- Menyimpan sejarah error (MSE) pada setiap epoch untuk dianalisis atau divisualisasikan.

model.predict(X_normalized):

- Menghasilkan prediksi jaringan neural berdasarkan input **X_normalized**.

scaler_y.inverse_transform(predictions_normalized):

- Mengembalikan prediksi ke skala asli menggunakan scaler target (**scaler_y**).
- Hasilnya (**predictions**) adalah prediksi dalam skala yang sama dengan target asli (**y**).

```python
# Convert to DataFrame
df = pd.DataFrame(data)

# Split features and target
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values.reshape(-1, 1)

# Normalize data
scaler_X = MinMaxScaler(feature_range=(0.1, 0.8))
scaler_y = MinMaxScaler(feature_range=(0.1, 0.8))

X_normalized = scaler_X.fit_transform(X)
y_normalized = scaler_y.fit_transform(y)

# Create and train model
model = BackpropagationNN(input_size=4, hidden_size=2, output_size=1)
history = model.train(X_normalized, y_normalized, epochs=5000, target_mse=0.00001)

# Predictions and metrics
predictions_normalized = model.predict(X_normalized)
predictions = scaler_y.inverse_transform(predictions_normalized)
mse = np.mean(np.square(y - predictions))
rmse = np.sqrt(mse)

# Accuracy calculation
predicted_classes = [1 if pred >= 0.5 else 0 for pred in predictions.flatten()]
actual_classes = y.flatten()
correct_predictions = np.sum(np.array(predicted_classes) == np.array(actual_classes))
accuracy = (correct_predictions / len(y)) * 100
```

np.mean(np.square(y - predictions)):

- **Mean Squared Error (MSE):**
- Menghitung rata-rata dari kuadrat selisih antara target asli (y) dan prediksi (predictions).
- Semakin kecil nilai MSE, semakin baik model.

np.sqrt(mse):

- **Root Mean Squared Error (RMSE):**
- Akar kuadrat dari MSE.
- Nilai ini lebih mudah diinterpretasikan karena dalam skala yang sama dengan target.

predicted_classes:

- Mengubah nilai prediksi menjadi kelas (0 atau 1) berdasarkan ambang batas 0.5.
- Jika prediksi ≥0.5\geq 0.5≥0.5, dianggap kelas 1; jika lebih kecil, kelas 0.

actual_classes:

- Mengubah target asli (y) menjadi array satu dimensi untuk perbandingan.

correct_predictions:

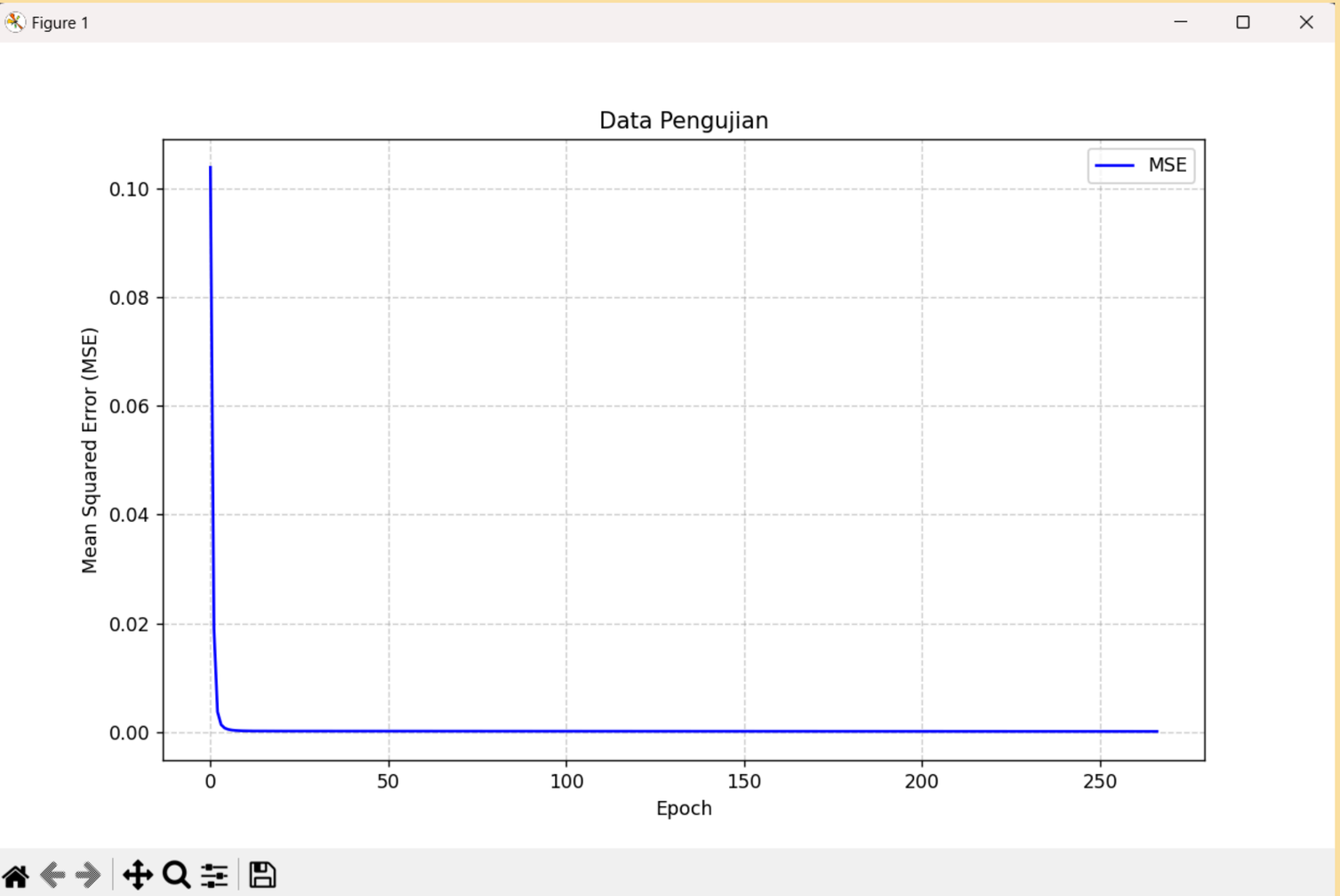
- Menghitung jumlah prediksi yang benar dengan membandingkan predicted_classes dan actual_classes.

accuracy:

- Menghitung persentase prediksi yang benar

$$\text{Akurasi} = \frac{\text{Jumlah Prediksi Benar}}{\text{Total Data}} \times 100$$

GRAFIK DATA PENGUJIAN



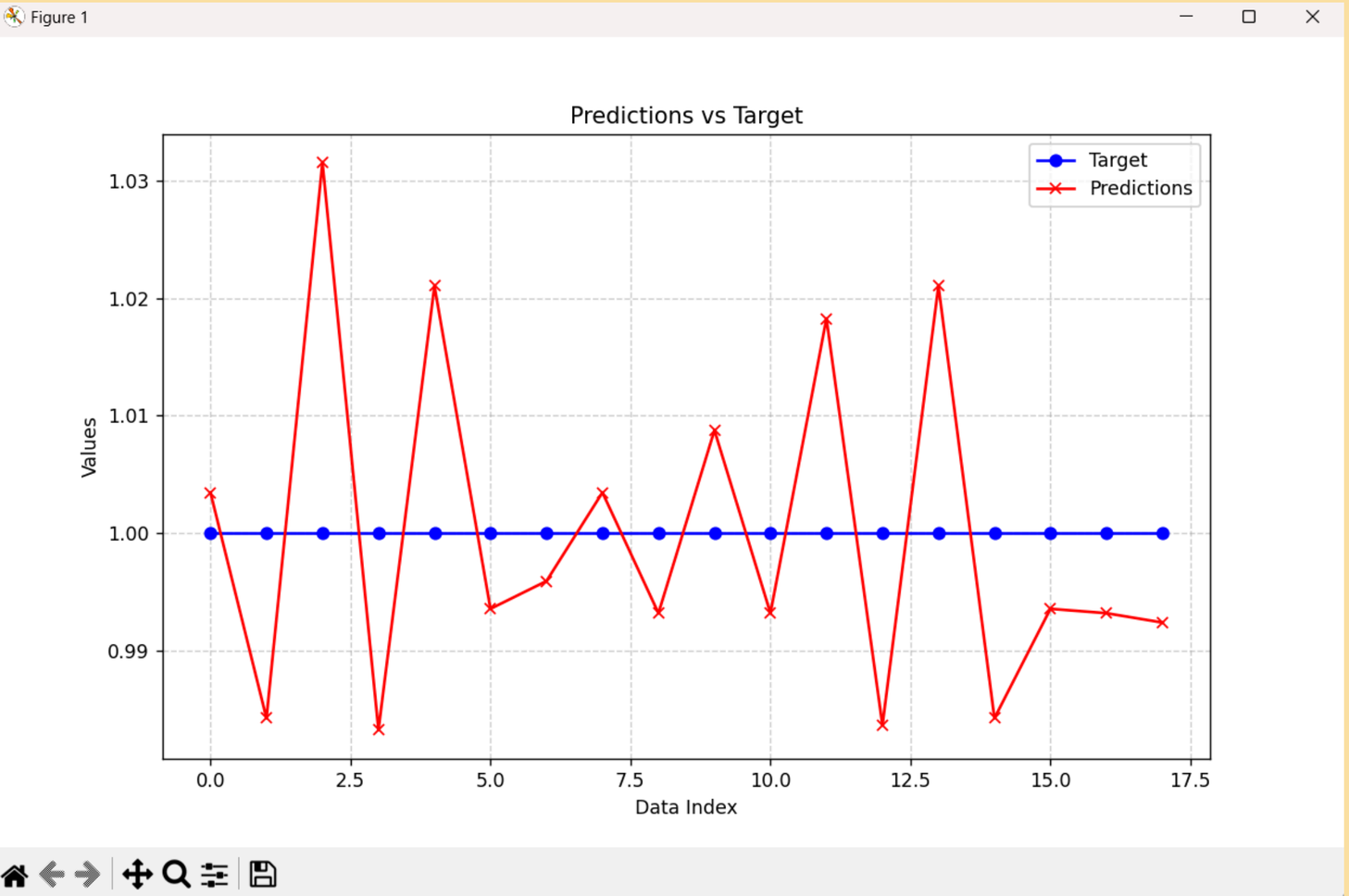
Penurunan Cepat di Awal

- Nilai Mean Squared Error (MSE) menurun drastis pada epoch awal, menunjukkan bahwa model belajar dengan cepat dan mampu memahami pola dasar data.

Stabilisasi di Akhir

- Setelah sekitar 20-50 epoch, MSE mendekati 0 dan grafik menjadi datar, menandakan model telah mencapai konvergensi dan mampu memprediksi dengan akurasi tinggi.

GRAFIK PERBANDINGAN ANTARA PREDIKSI DENGAN TARGET



Prediksi Berfluktuasi

- Nilai prediksi (garis merah) menunjukkan fluktuasi signifikan di sekitar nilai target (garis biru) yang konstan, menandakan bahwa model belum sepenuhnya stabil dalam memprediksi data.

Error Masih Terlihat

- Meskipun sebagian prediksi mendekati target, variasi yang tinggi menunjukkan ketidaksesuaian antara prediksi dan target, yang mengindikasikan model masih perlu penyempurnaan.

KESIMPULAN DATA

```
Epoch 0, MSE: 0.103862
Epoch 100, MSE: 0.000137
Epoch 200, MSE: 0.000113
Target MSE reached at epoch 266
HASIL PENGUJIAN DAN PREDIKSI
```

NIS	Prediksi	Target	Kategori	JST (Error)	Hasil
1542	1.0035	1.0	Memahami	0.00345	Benar
1543	0.9842	1.0	Cukup Memahami	0.01577	Benar
1544	1.0316	1.0	Cukup Memahami	0.03155	Benar
1545	0.9832	1.0	Cukup Memahami	0.01677	Benar
1546	1.0210	1.0	Cukup Memahami	0.02105	Benar
1547	0.9935	1.0	Memahami	0.00645	Benar
1548	0.9959	1.0	Memahami	0.00409	Benar
1549	1.0035	1.0	Memahami	0.00345	Benar
1550	0.9932	1.0	Memahami	0.00682	Benar
1551	1.0088	1.0	Memahami	0.00875	Benar
1552	0.9932	1.0	Memahami	0.00682	Benar
1553	1.0183	1.0	Cukup Memahami	0.01827	Benar
1554	0.9836	1.0	Cukup Memahami	0.01637	Benar
1555	1.0210	1.0	Cukup Memahami	0.02105	Benar
1556	0.9842	1.0	Cukup Memahami	0.01577	Benar
1557	0.9935	1.0	Memahami	0.00645	Benar
1558	0.9932	1.0	Memahami	0.00682	Benar
1559	0.9924	1.0	Memahami	0.00762	Benar

```
HASIL AKHIR:
MSE (Mean Squared Error): 0.000203
RMSE: 0.014261
Akurasi: 100.00%
```

Akurasi Tinggi

- Model mencapai akurasi 100% dengan MSE sebesar 0.000023 dan RMSE 0.014261, menunjukkan prediksi sangat mendekati target.

Error Kecil dan Konsisten

- Selisih antara prediksi dan target (JST Error) berkisar antara 0.00345 hingga 0.03155, dengan sebagian besar kategori "Memahami" dan "Cukup Memahami", menandakan performa model sangat baik.

REFERENSI

- Sakinah, N. P., Cholissodin, I., & Widodo, A. W. (2018). Prediksi Jumlah Permintaan Koran Menggunakan Metode Jaringan Syaraf Tiruan Backpropagation. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, 2(7), 2612-2618.
- Solikhun, Safii, M., & Trisno, Agus. (2017). Jaringan Saraf Tiruan untuk Memprediksi Tingkat Pemahaman Siswa terhadap Mata Pelajaran dengan Menggunakan Algoritma Backpropagation. *Jurnal Sains Komputer & Informatika (J-SAKTI)*, 1(1), 24-36.

TERIMA KASIH